1) Integer Type - This type stores positive or negative numbers, whole numbers without any decimal values.

```
var num : Int = 7
```

Floating Point Type - This type stores numbers with fractional part. There are 2 categories based on number of decimal digit capacity: Float & Double.

```
var num1 : Double = 7.77
```

Boolean Type - This type stores the value in True or false format. Either it is true or it is false.

```
var st : Boolean = true
```

Strings type- This type of data stores the sequence of character values. The data is written in " ".

var str: String = "Student"

2) Data Classes- It is a simple class which is used to hold state/data and contains standard functionality. "data" keyword is used to declare a class as data class.

For declaring, a data class must contain atleast one primary constructor with property argument. Data class internally contains following functions.

1) equals() : Boolean
2) hashCode() : Int
3) toString() : String
4) component()
5) copy()

(Compiler automatically derives this func^n.)

```
data class laptop (var brand: String, var price
                                          : Int)
{
  fun show()
  {
    println("laptop")
  }
}
```

```
fun main()
{
  var x = laptop("HP", 2000)
  println(x)
  var y = laptop("Dell", 2500)
  println(y)
  var z = x.copy()      //copy func^n
  println(z)
  println(x.equals(y))  //equals func^n
}
```

3) Null Safety is a procedure to eliminate the risk of null reference from the source code. Kotlin compiler throws NullPointerException immediately if it finds the null argument without executing other statements.

Nullable Types: They are declared by putting ? after the data type of the variable.

eg:

```
var str : String ? = "student"
str = null      // will work also now.
print(str)         it will print null if we printed
                                                 it.
```

Non Nullable Types - Here the compiler will throw an error.

```
var str: String = "student"
str = null    // compiler throws error
print(str)      because we have not used ?
```

To check for null reference if can be used. Kotlin's if expression is used for checking the null condition and returns value.

```
fun main()
{
   var str : String ? = null
   println (str)

    var l1 = if (str != null){
             str.length } else { -1 }

    println(l1)
}
```

```
null
-1
```

4) Safe Call Operator - Kotlin has safe call operator (?.) which executes the line of code only when the specific reference holds a non-null value.

- It means that it will proceed ahead and execute only and only if it (the reference) is not null if value is null it will directly return null without checking ?
- Whatever lines of code we want to execute, it will execute only if our reference is not null.

anyth-ing

Page No.:
Date : / /

→ Elvis Operator - Kotlin has elvis operator (?:). It is used to return the not null value even the expression or reference is null. It is used to check the null safety of values.

- If we have a reference that holds null then using elvis operator it can return a non null value or it will return whatever value we assign it manually.

→ Not Null Assertion Operator - In Kotlin, not null assertion operator is used when we are certain that a variable or expression is not null. It converts any value of nullable type to a non null type and throws 'Kotlin Null Pointer Exception' if the value is actually null.

- If the value is not null it will work as per the regular manner but if the value is null it will throw nullpointer exception.

- We should use this operator only when we are absolutely sure that the value is not null and we want to treat it as a non-nullable type.

- Instead of using this operator it is better to use the alternatives ie. safe call (?.) and elvis (?:) operators.

```
fun main()
{
   var str : String ? = null
   if (str != null)
       println( str.length)
   println( str ?. length) // safe call
   var value1 = str ?: "NA" // elvis
   println (value1)
   var value2 = str !!. length // Not Null
                                 assertion
   println (value2)
}
```

Output

null //as safe call returns null
NA //elvis assign NA to our variable
Exception occurs which is NullPointer Exception
and further execution stops. compiler
throws error.

5) Constructor is a function without name which is used to initialize objects of class.
There are 2 types: Primary.
Secondary.

→ Primary constructor is a part of class header It can take parameters and is defined after classname using constructor keyword. (optional)

```
class abc constructor (n: string="student")
{
  var str: String=n
  println (n)
}
fun main()
{
  var x = abc("dev")
}.
```

→ Secondary Constructor is defined inside the class and allow defining it with logic. which allowes for more flexible object creation.

```
class abc constructor (n: String)
{
  var str1 : String=" "
  var a : Int = 0
  constructor(a: Int, str1: String): this(str1)
  {
      this.a = a
      this.str1 = str1
  }
fun show( )
{
  println ("$str1 and $a")
 }
}
fun main( )
{
 var x = abc (17, "shyam")
  x.show()
}
```

6) Extension function - It provides us with a facility to add methods to a class without inheriting a class or using extending it.

```
class sample {
 var skills: string = " "
  fun show ()
  {
    println (skills)
   }
}
fun sample.plus ( v1 : sample) : sample
{
  var v2 = sample()

  v2.skills = this.skills + " " + v1.skills

  return v2
}
```

Extension function enhances flexibility and extending functionality of classes without modifying source code.

```
fun main()
{
 var s1 = sample()
 s1.skills = "kotlin"
 println(s1.skills)

  var s2 = sample()
  s2.skills = "java"
  println (s2.skills)

  var s3 = s1.plus(s2)
  println (s3.skills)
}
```

7) Companion Object - It returns an object without even creating an object or instance of class. when we want to use an object without creating one we can use companion object. It is declared inside the class. To call it we have to use classname. function_name ().

```
class abc
{                                    fun main ()
  companion object                   {
  {                                      abc.show()
   fun show()                        }
   {
     println ("Hello")
   }
  }
}
```

It works similiar to static in java but. To access actual properties of static we need to write @ JumStatic to make it actually static.

```
class abc
{
  companion object
  {
    @JumStatic
    fun show()
    {
      println("Hello")
    }
  }
}
fun main()
{
  abc.show()
}

public class sample
{
  public static void main(String args[])
  {
    abc.show()
  }
}
```

used for : 1) Factory Design Pattern.
2) Encapsulation

8) In kotlin , when is used to access the functionality of switch in java. A certain block of code will be executed when some condition is fulfilled. One by one the target is matched until found and terminates after the match and exitsthe block of code. It can be used as a statement and used as an expression.

```
fun main ( )
{
  var n : Int = 2
  when (n)
  {
    1 → print ("One")
    2 → print ("Two")
    3 → print ("Three")
    4 → print ("Four")
    else → print ("Invalid")
  }
}
```

```
Two
```

and

```
fun main()
{
  var n : Int = 2
  var str = when (n) {
            1 → "one"
            2 → "two"
            3 → "three"
            else → "invalid"
  }
  print (str)
}
```

```
Two
```

9) 1) Kotlin is way more concise than Java.

2) Our code becomes easier to maintain. Some features for code conciseness: data class, type interface. Kotlin requires less code to achieve same functionality.

3) Null Safety - Kotlin has built in Null Safety into type system, helping to avoid the common null pointer exception by making ~~mutat~~ nullable and non nullable types explicit.

4) Extension function - adding func.^{n} to existing classes with new functionality without modifying their source code.

5) Immutability - Encouragement of immutable data structures. val - immutable variables.

6) functional programming features - Higher order functions and lambdas can lead to more expressive and concise code.

10) Recursion - Function calling itself to solve problem until the termination / base condition is met. A major problem is broken into smaller sub problems.

```
fun main()
{
  var n=5
  factorial(n)
}
```

```
fun factorial(num: Int): Int
{
  if(num==0)
    return 1
  else
    return num * factorial(num-1)
}
```

Tail Recursion - In some cases, recursion can lead to deep full stack problem and stack overflow issues. The call to sub problems continues and all the calculations are in pending state. So to avoid that we have to explicitly inform compiler to use tail recursion concept where the calculation is done simultaneously and not in pending state. works for even larger inputs.

```
tailrec fact (num: BigInteger, res: BigInteger)
{                                          : BigInteger
  if(num == BigInteger.ZERO)
     return res
  else
     return fact(num - BigInteger.ONE, num*res)
}
```

```
fun main ()
{
  var num = BigInteger ("5")
  print ( fact (num, BigInteger.ONE))
}
```

1) Anonymous Innor Class allows you to creat an object of a class with certain modifiers without having to actually subclass or name that class.

It is basically used with Interface or abstract class that have to be implemented or overrides for specific action.

```
interface asd
{
  fun show()
}
fun main()
{
  var a = object : asd {

    override fun show(){
    println("in show method")
    }

  a.show()
}

abstract class shape{
  abstract fun draw()
}
```

```
fun main()
{
  val c = object : shape(){
     override fun draw(){
     println("drawing method")
     }
  }

   c.draw()
}
```